STL

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń, lukasz@coders.school
Kamil Szatkowski, kamil.szatkowski@nokia.com

Łukasz Ziobroń

Not only programming experience:

- C++ and Python developer @ Nokia & Credit Suisse
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Webmaster (HTML, PHP, CSS) @ StarCraft Area

Training experience:

- C++ trainings @ Coders School
- Practial Aspects Of Software Engineering @ PWr, UWr
- Nokia Academy @ Nokia
- Internal corporate trainings

Public speaking experience:

- Academic Championships in Team Programming
- code::dive conference
- code::dive community

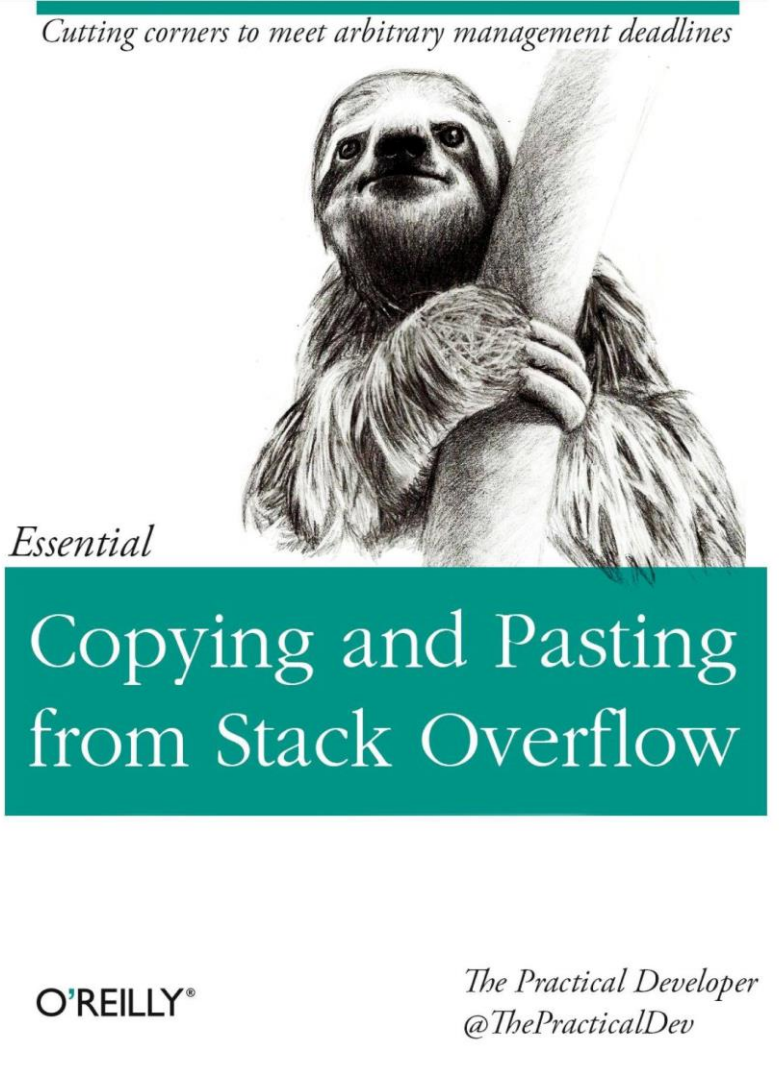
SODD

How to deal with a problem in programming?

- Waste some time trying to fix compiler errors
- Google the problem
- Open the first link (probably StackOverflow)
- Copy and paste the solution

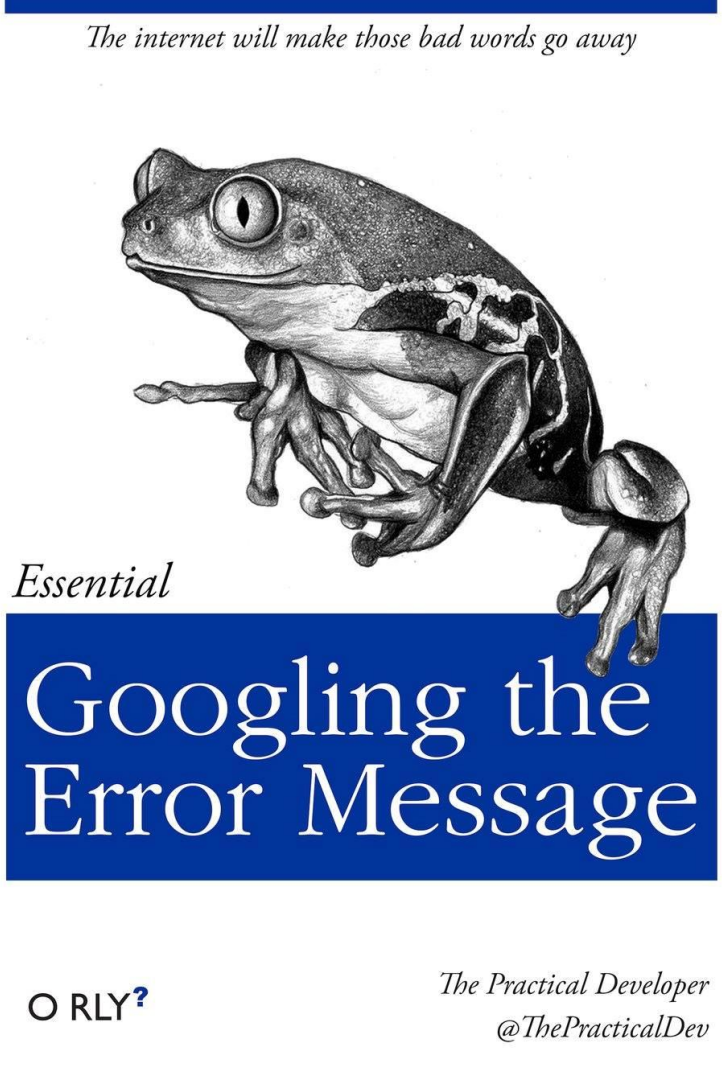
SODD

StackOverflow Driven Development



SODD

Google
Driven Development?



Agenda

1. **Containers**
2. Iterators
3. Functors
4. Algorithms

Containers

Traits

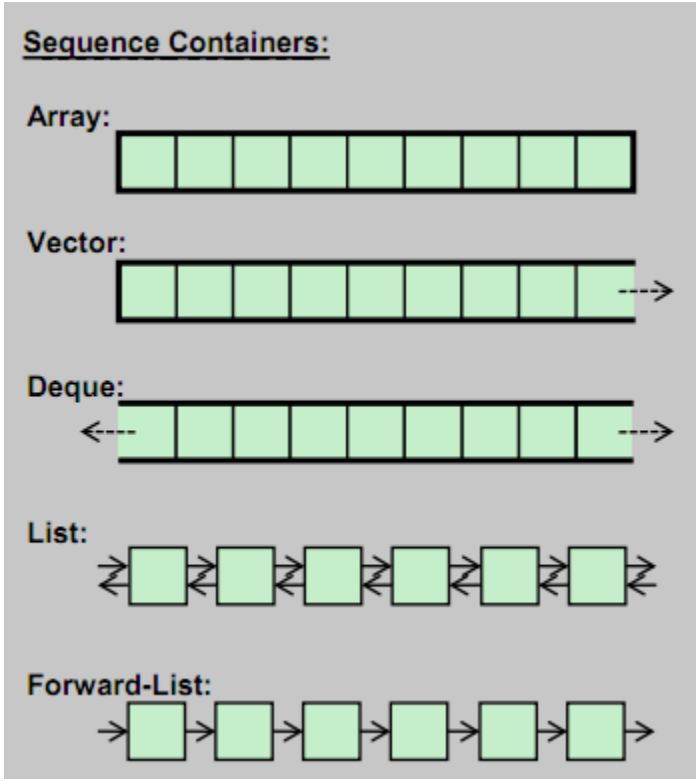
- generic (based on templates)
- own objects
- manage memory of objects
- provide access to objects (directly or via iterators)

Container families

- Sequence containers
- Associative containers
- Adaptors
- Other containers

Sequence containers

- std::array<>
- std::vector<>
- std::deque<>
- std::list<>
- std::forward_list<>



Sequence containers

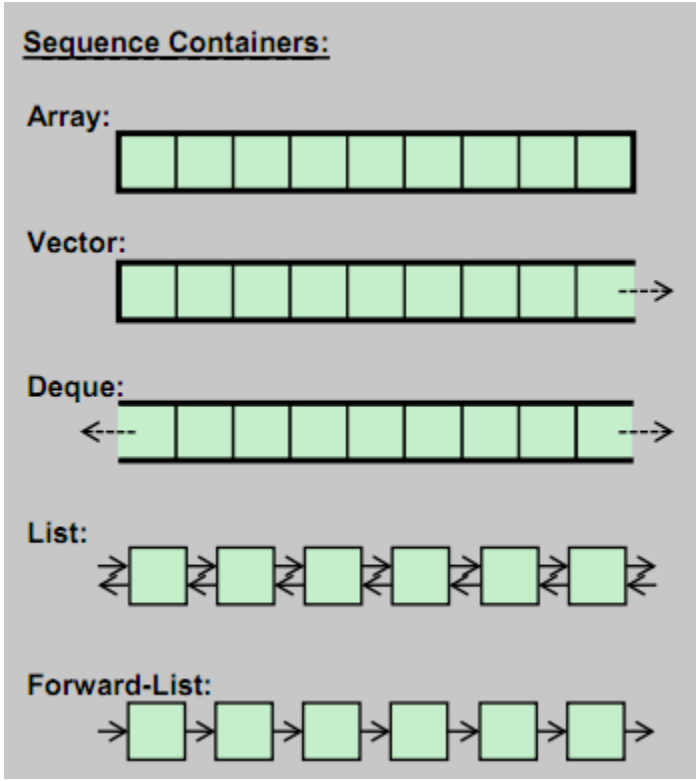
Base operations

- begin(), end(), *rbegin(), rend()*
- cbegin(), cend(), *crbegin(), crend()*
- *size()*, max_size(), empty(),
- *resize(),*
- front(), *back(),*
- *assign()*, *emplace()*, *insert()*, *erase()*,
- swap(), *clear()*

std::array<>

Traits

- STL equivalent to **Type a[]**
- contiguous storage on stack (*data()*)
- random access - *O(1)*
- fixed-size aggregate
- pure data, no hidden fields
- cache-friendly



std::array<>

Example:

```
std::array<int, 5> a = { 1, 2, 4, 5, 6}; // eq. int a[5] = { 1, 2, 4, 5, 6};
a[0] = 5;                                // a == { 5, 2, 4, 5, 6}
a.at(1) = 7;                             // a == { 5, 7, 4, 5, 6}
a[4] = a.front();                        // a == { 5, 2, 4, 5, 5}
a.fill(5);                               // a == { 5, 5, 5, 5, 5}

std::cout << a.size()                    // 5
std::cout << a.max_size()                // 5

std::array<int, 5> b = { 5, 6, 7, 8, 9};
b.swap(a); // a == { 5, 6, 7, 8, 9} , b == { 5, 5, 5, 5, 5}
```

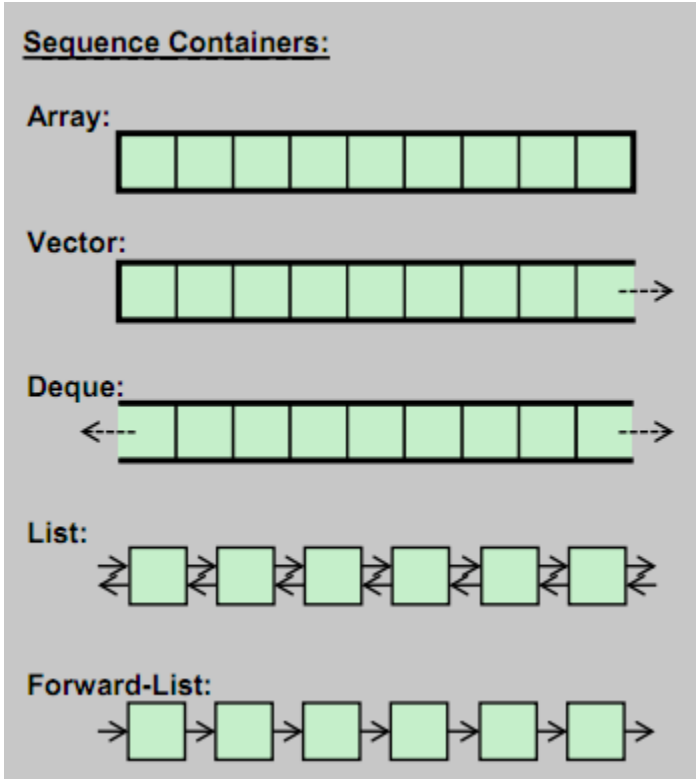
std::array<>

Excercise

1. Create std::array with size 10.
2. Fill it with numbers 5.
3. Assign to the 4-th element value 3.
4. Create another std::array with the same size.
5. Swap arrays.
6. Get a pointer to the raw data
7. Print both arrays – one array in one line.

Sequence containers

- std::array<>
- **std::vector<>**
- std::deque<>
- std::list<>
- std::forward_list<>



std::vector<>

Traits

- storage **dynamically allocated on heap**
- contiguous storage (*data()*)
- random access - *O(1)*
- resizeable
- cache-friendly
- insertion at the end is provided with amortized constant time - *O(1)*

std::vector<>

Additional methods

- resize(), shrink_to_fit()
- capacity()
- reserve()
- push_back(), pop_back(), emplace_back()
- data()

std::vector<>

Excercise

After each step print the vector's size, capacity and max_size.

1. Create a vector with the following values { 1, 2, 4, 5, 6 }.
2. Create 12 in the vector at the beginning using emplace().
3. Resize vector to size 10 and fill it with 5.
4. Reserve space for 20 elements.
5. Shrink to fit.
6. Clear the vector

std::vector<bool>

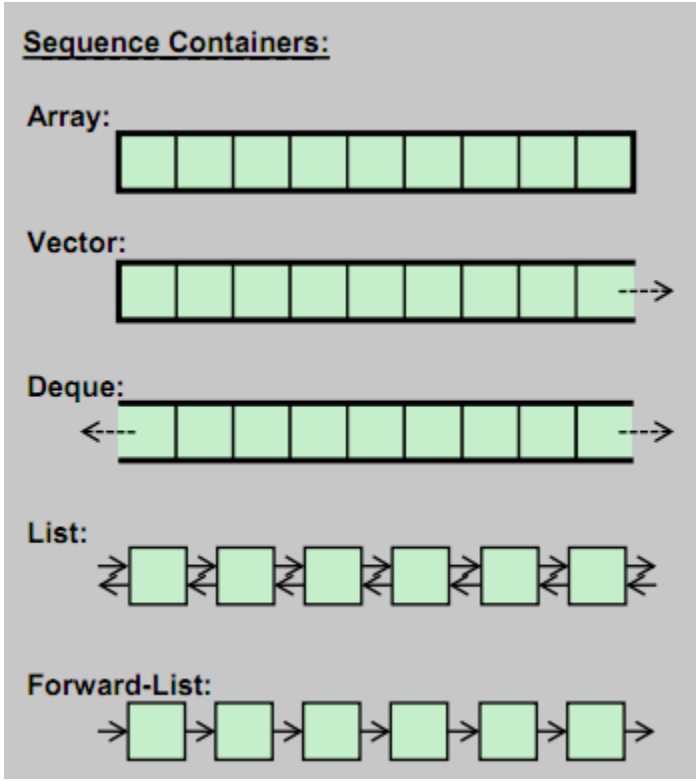
Traits

- specialization optimized for space (like bitset but dynamic)
- elements are not constructed using alocator object
- special proxy type class is used for accessing the value
- pointers and iterators are not intuitive

If dynamic allocation is not needed std::bitset is a better choice

Sequence containers

- std::array<>
- std::vector<>
- **std::deque<>**
- std::list<>
- std::forward_list<>



std::deque<>

Traits

- similar to vector
- storage **dynamically allocated on heap**
- insertion at the beginning and end is provided with amortized constant time - *O(1)*
- random access - *O(1)*
- non-continuous storage

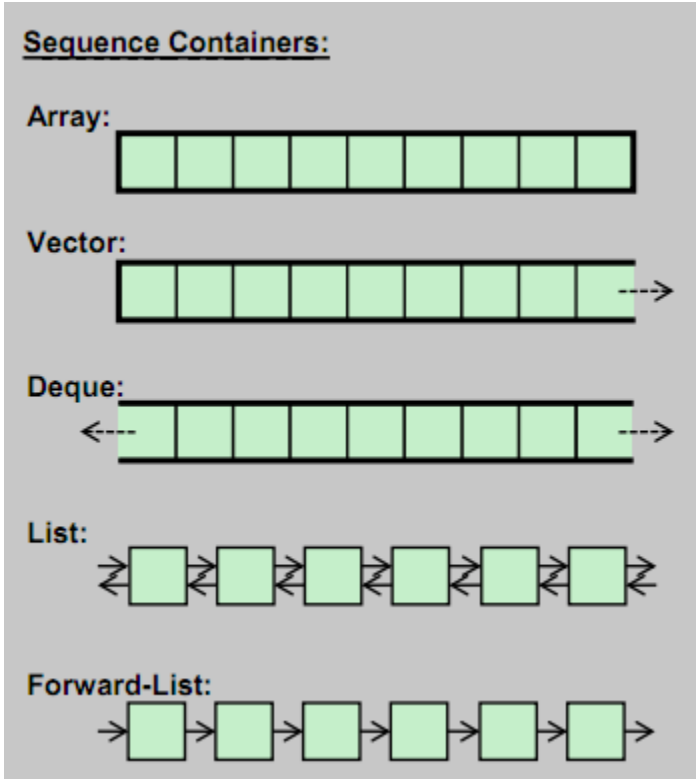
std::deque<>

Additional methods

- shrink_to_fit()
- push_back(), pop_back(), emplace_back()
- push_front(), pop_front(), emplace_front()

Sequence containers

- std::array<>
- std::vector<>
- std::deque<>
- **std::list<>**
- std::forward_list<>



std::list<>

Traits

- bidirectional access - *O(N)*
- storage **dynamically allocated on heap**
- double-linked list
- constant time insertions and deletions - *O(1)*
- cache inefficient
- iterators are not invalidated

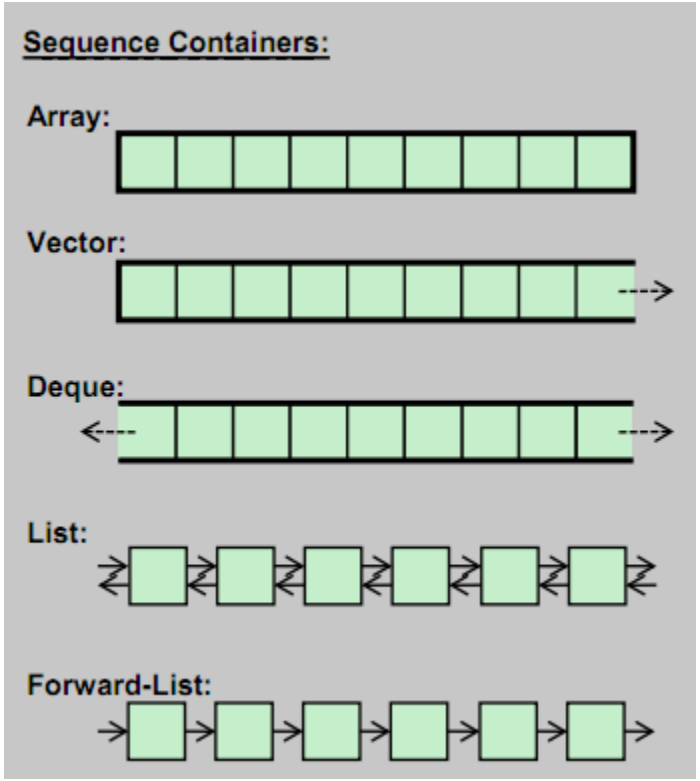
std::list<>

Additional methods

- push_back(), pop_back(), emplace_back()
- push_front(), pop_front(), emplace_front()
- splice(), unique(), merge(), sort(), reverse()
- remove(), remove_if()

Sequence containers

- std::array<>
- std::vector<>
- std::deque<>
- std::list<>
- **std::forward_list<>**



std::forward_list<>

Traits

- forward access - *O(N)*
- storage **dynamically allocated on heap**
- single-linked list
- fast insertions and deletions - *O(1)*
- cache inefficient
- more efficient than *std::list* (processing and memory)

std::forward_list<>

Additional methods

- insert_after(), emplace_after(), erase_after()
- push_front(), pop_front(), emplace_front()
- splice(), unique(), merge(), sort(), reverse()
- remove(), remove_if()

std::forward_list<>

Methods missing

- rbegin(), rend()
- crbegin(), crend(), size(), back()

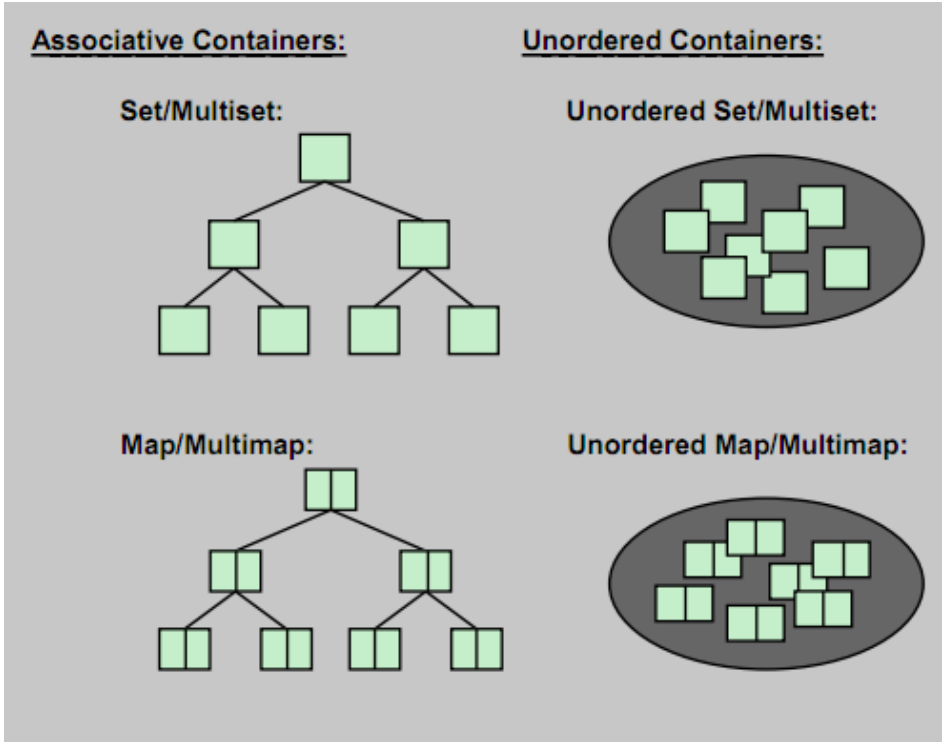
Associative containers

Ordered

- set
- multiset
- map
- multimap

Unordered

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap



Ordered associative containers

Traits

- storage dynamically allocated on heap
- support bidirectional iterators
- sorting done by default with std::less
- all elements are always const
- typically implemented with binary search tree

Unordered associative containers

Traits

- storage **dynamically allocated on heap**
- support forward iterators
- all elements are always const
- fast access to elements (hashing containers)
- require specialized hash() function for uncommon objects
- organized into buckets

Unordered associative containers

Bucket and hash policy interface methods

- bucket(), bucket_count(), bucket_size(), max_bucket_count()
- rehash(), load_factor(), max_load_factor()

Ordered associative containers

Excercise

1. Create std::map of integers to strings with the following content:
   {1 → ‘one’, 2 → ‘two’, 3 → ‘thr’, 4 → ‘four’, 5 → ‘five’}
2. Add a new pair: 3 → ‘three’
3. Erase element with key 5.
4. Find element with key 4 and print it’s key and value.

Sequence and associative containers - summary

- std::array<T, N> - stack allocation
- std::vector<T> - heap allocation
- std::deque<T> - heap allocation
- std::list<T> - heap allocation
- std::forward_list<T> - heap allocation
- std::set<T> - heap allocation
- std::map<T> - heap allocation
- std::unordered_set<T> - heap allocation
- std::unordered_map<T> - heap allocation

Adaptors

- std::stack<>
- std::queue<>
- std::priority_queue<>

Other containers

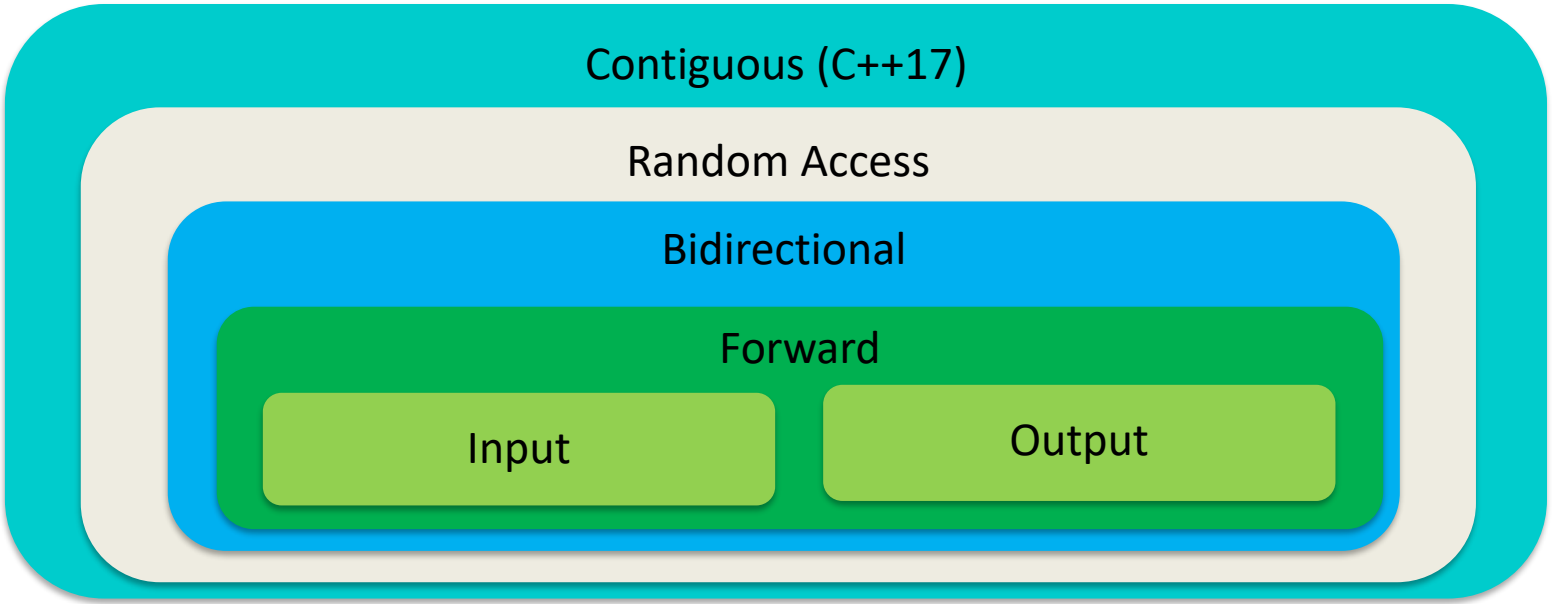
- std::string / std::wstring – heap allocation
- std::valarray – stack allocation
- std::tuple – stack allocation
- std::bitset – stack allocation

Agenda

1. Containers
2. **Iterators**
3. Functors
4. Algorithms

Iterators

The *iterator* is any object that points to some element in a range of elements, has the ability to iterate through the elements of that range using a set of operators (at least increment (++) and dereference (*) operator)



Iterators

Base operations

- copy-constructible, copy-assignable, destructible
- can be incremented (*++a, a++*)

Input iterators

Supports sequential input operations where each value pointed by the iterator is read only once and then the iterator is incremented

Operations

- can be dereferenced as an rvalue (if in proper state)
- can be incremented (if in proper state)
- can be compared (eq or neq) with another iterator

Examples

- std::istream_iterator

Output iterators

Supports sequential output operations where a value is written to the element pointed by the iterator and then the iterator is incremented

Operations

- can be dereferenced as an lvalue (if in proper state)
- can be incremented (if in proper state)
- reading value is inadvisable

Examples:

- std::ostream_iterator

Forward iterators

Iterator that can be used to access the sequence of elements in range in the direction that goes from its beginning towards its end

Operations

- aggregates input and output iterators
- can be constructed with default constructor
- supports multipass

Examples

- std::forward_list::iterator
- std::unordered_set::iterator
- std::unordered_map::iterator

Bidirectional iterators

Iterator that can be used to access the sequence of elements in a range in both directions.

Operations

- aggregates forward iterator
- can be decremented

Examples

- std::list::iterator
- std::map::iterator
- std::set::iterator

Random Access Iterators

Iterators that can be used to access elements at an arbitrary offset position relative to the element they point to, offering the same functionality as pointers

Operations

- aggregates bidirectional iterator
- support arithmetic operators
- can be compared with inequality relational operators
- supports the offset dereference operator ([])

Examples

- std::vector::iterator
- std::deque::iterator
- std::array::iterator

Iterators

Additional operations

- advance()
- distance()
- begin(), end()
- prev(), next()

Agenda

1. Containers
2. Iterators
3. **Functors**
4. Algorithms

Functions and functors

Many of STL algorithms requires additional parameters with predicate, comparator or other function

```
bool is_odd(int a)
{
   return (a % 2) != 0;
}

std::vector<int> a = { 1, 2, 3, 4, 5 };
std::find_if(a.begin(), a.end(), is_odd);
```

Functions and functors

Instead of providing function adres functor object can be used.

```
class Is_odd {
  bool operator() (int a)
  {
       return (a % 2) != 0;
  }
};

std::vector<int> a = { 1, 2, 3, 4, 5 };
std::find_if(a.begin(), a.end(), Is_odd());
```

Lambda functions

```
[](){}                        // empty lambda, does nothing
[](){ return 4; }             // unnamed lambda returning 4
[](int i){ return i >= 0 }  // unnamed lambda returning if parameter is >= 0

auto multiplyByTen = [](int k){ return k * 10 };  // named lambda
int number = multiplyByTen(5);                    // number = 50
```

Lambda functions

```
int a {5};
auto add5 = [=](int x) { return x + a; };


std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int even_count = 0;
for_each(v.begin(), v.end(), [&even_count] (int n)
{
    cout << n;
    if (n % 2 == 0)
        ++even_count;
});
cout << "There are " << even_count << " even numbers in the vector." << endl;
```

Lambda functions – capture list

Inside brackets [] we can include elements that the lambda should capture from the scope in which it is create. Also the way how they are captured can be specified.

[ ] empty brackets means that inside the lambda no variable from outer scope can be used.

[&] capture by reference, including this pointer. Functor created by lambda expression can read and write to any captured variable and all of them are kept inside lambda by reference.

[=] captured by value, including this pointer. All variables from outer scope are copied to lambda expression and can be read (writing is not allowed – const is added)

[a, b, &] captures variable a and b by copy and any other by reference.

[*this] (C++17) captures this pointer by value. Anyway, this is implicitly captured by [&] and [=].

Generic lambda (C++14)

In C++11 parameters of lambda expression must be declared with use of specific type. C++14 allows to declare paramaters as *auto* (*generic lambda*).

This allows compiler to deduce the type of lambda parameter in the same way as parameters of templates are deduced. In result compiler generates code equivalent to closure class given below.

```
auto lambda = [](auto x, auto y) { return x + y; }

struct UnnamedClosureClass  // code generated by the compiler for above 1-liner
{
    template <typename T1, typename T2>
    auto operator()(T1 x, T2 y) const
    {
        return x + y;
    }
};
auto lambda = UnnamedClosureClass();
```

std::function

std::function can hold a functional object. It does the same thing as a pointer to function. However, you cannot have a pointer to the lambda that has something on a capture list. std::function solves this problem.

```
void print_num(int i) {
    std::cout << i << '\n';
}

// store a free function
std::function<void(int)> display = print_num;
display(-9);

// store a lambda
int a = 10;
std::function<void()> lambdaDisplay = [a](){ print_num(42 + a); };
lambdaDisplay();
```

Functions and functors

Excercise

1. Create std::array of 6 doubles with the following elements
   {5.0, 4.0, -1.4, 7.9, -8.22, 0.4}
2. Sort elements using std::sort and provide functor, that sorts by absolute values (std::abs)
3. Change functor object to lambda function.

Agenda

1. Containers
2. Iterators
3. Functors
4. **Algorithms**

Algorithms

STL algorithms is set of functions that operate on range defined by iterators

Example of usage:

```
std::vector<int> a = {1, 2, 3, 4, 5};

std::transform(a.begin(),
               a.end(),
               std::ostream_iterator<int>(std::cout, " "),
               [](const auto & a){ return a * 25; });

// Result:
// 25 50 75 100 125
```

Algorithms - categories

- Non-modifying sequence operations
- Modifying sequence operations
- Sorting
- Partitions
- Binary search
- Merge
- Heap
- Min/max
- Other
- Numeric

Non-modyfing sequence operators

- std::all_of, std::any_of, std::none_of
- std::for_each
- std::find, std::find_if, std::find_if_not, std::find_end, std::find_first_of, std::adjacent_find
- std::count, std::count_if
- std::mismatch
- std::equal
- std::is_permutation
- std::search, std::search_n

Non-modyfing sequence operators

Excercise

1. Write function *is_palindrome* that will check if given std::string is a palindrome or not. Use std::mismatch().

Modifying sequence operations

- std::copy, std::copy_n, std::copy_if, std::copy_backward
- std::move, std::move_backward
- std::swap, std::swap_ranges, std::iter_swap
- std::transform
- std::replace, std::replace_if, std::replace_copy, std::replace_copy_if
- std::fill, std::fill_n
- std::generate, std::generate_n
- std::remove, std::remove_if, std::remove_copy, std::remove_copy_if
- std::reverse, std::reverse_copy
- std::rotate, std::rotate_copy
- std::shuffle, std::random_shuffle

Sorting

- std::sort
- std::stable_sort
- std::partial_sort, std::partial_sort_copy
- std::is_sorted, std::is_sorted_until
- std::nth_element

Sorting

Excercise

1. Create empty std::deque for int values.
2. Generate 14 values using std::back_inserter and std::generate_n with rand() but limited to 7.
3. Sort values and print.
4. Leave only unique values in container and print them.
5. Rotate them around the middle element and print result.

Partitions

- std::is_partitioned
- std:: partition
- std::stable_partition
- std::partition_copy
- std::partition_point

Binary search

- std::lower_bound
- std::upper_bound
- std::equal_range
- std::binary_search

Merge

- std::merge
- std::inplace_merge
- std::includes
- std::set_union
- std::set_intersection
- std::set_difference
- std::set_symetric_difference

Heap

- std::push_heap
- std::pop_heap
- std::make_heap
- std::sort_heap
- std::is_heap
- std::is_heap_until

Min/max

- std::min
- std::max
- std::minmax
- std::min_element
- std::max_element
- std::minmax_element

Other

- std::lexicographical_compare
- std::next_permutation
- std::prev_permutation

Numeric

- std::iota
- std::accumulate
- std::inner_product
- std::adjacent_difference
- std::partial_sum
- std::reduce
- std::exclusive_scan
- std::inclusive_scan
- std::transform_reduce
- std::transform_exclusive_scan
- std::transform_inclusive_scan

Group exercise

Excercise (2-4 people in a group)

Cryptographic application.

Requirements:

1. Substitution ciphering (map letter -> cipher)
2. Encryption and decryption
3. Cipher is generated randomly
4. Input data: cin and/or file
5. Output data: cout and/or file

`Use as much STL as possible ☺`

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń, lukasz@coders.school
Kamil Szatkowski, kamil.Szatkowski@nokia.com